EUROPEAN PATENT OFFICE

Patent Abstracts of Japan

PUBLICATION NUMBER : 07281682
PUBLICATION DATE : 27-10-95

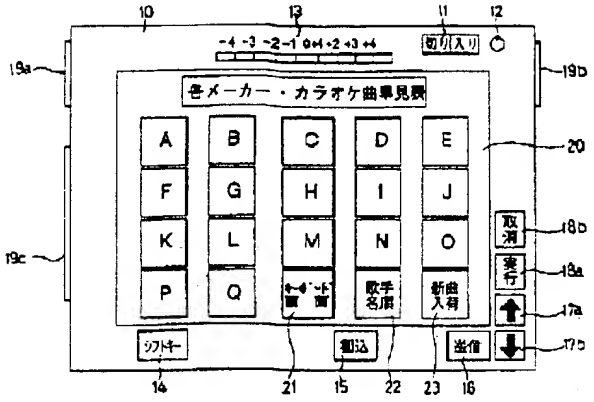
APPLICATION DATE : 11-04-94
APPLICATION NUMBER : 06071888

APPLICANT : YUASA NAGAO;

INVENTOR : YUASA NAGAO;

INT.CL. : G10K 15/04 G11B 27/34

TITLE : KARAOKE MUSIC SELECTION SYSTEM



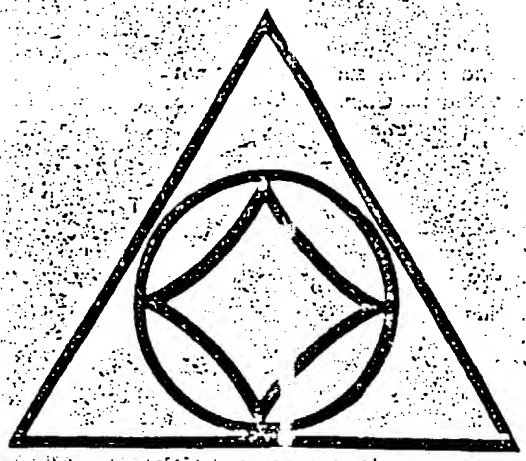
ABSTRACT : PURPOSE: To automatically select KARAOKE music to be played.
CONSTITUTION: The KARAOKE music selection system is equipped with a KARAOKE maker display means which displays a list of KARAOKE makers A-Q on the touch screen 20 of a KARAOKE remote control unit, a KARAOKE music display means which displays lists of KARAOKE music by the KARAOKE makers selected from the list of the KARAOKE makers A-Q, a reserved music list means which reserves a performance of KARAOKE music selected from the list of KARAOKE music, and a transmitting means which sends the reserved KARAOKE music to a KARAOKE main body.

The DSS development system

by ROBERT H. BONCZEK
Purdue University
West Lafayette, Indiana
NASIR GHIASEDDIN
University of Notre Dame
Notre Dame, Indiana
CLYDE W. HOLSAPPLE
University of Illinois
Champaign, Illinois
and
ANDREW B. WHINSTON
Purdue University
West Lafayette, Indiana

ABSTRACT

As decision support systems become more commonplace, the demand for automatic and semiautomatic DSS development systems increases proportionately. Such systems provide a set of tools that guide the construction of models in response to a user's query. This paper describes a set of such tools that provide capabilities for analysis, design, module management, and report and graphics generation.



AFIPS

CONFERENCE PROCEEDINGS

VOLUME 55

1986

NATIONAL COMPUTER CONFERENCE

ADDIE MA TOX
Editor and Proc ram Chair
MARY L ICH
Conference Chair

Sponsored by
AMERICAN FEDERATION OF INFORMATION PROCESSING SOC ETIES, INC.
ASSOCIATION FOR COMP TING MACHINERY
DATA PROCESSING MANAGE MENT ASSOCIATION
IEEE COMPUTER SOCIETY
SOCIETY FOR COMPUT R SIMULATION

AFIPS PF ESS
1899 PRESTON WH ITE DRIVE
RESTON, VIRGIN IA 22091

June 16–19, 1986
Las Vegas, Nevada

INTRODUCTION

In recent years, the need for increased productivity in managerial decision-making activities has been felt both in private and in public sectors. This need is mainly motivated by the competitive nature of the business world, which calls for more and more efficiency as an essential ingredient for business survival. Decision support systems (DSS) have been shown to increase management's effectiveness and productivity in handling decision problems. The potential benefits of decision support systems have created an ever-increasing need for these systems. This need has accelerated the efforts to build more and more such systems. As the potential benefits of decision support systems are realized by more decision makers in various fields, the need for such systems will increase even more. In 1978 only 20% of all applications developed were for management control, planning, and analysis (which roughly falls into the area of DSS), while 80% were operational. However, since then this breakdown has changed dramatically and it is estimated[1] that by 1983, 55% of the new programs will be written for management control, planning, and analysis, and only 45% of the new applications will be operational. The foregoing discussion suggests that there is a serious need for many new decision support systems to aid decision makers in various fields.

Although the computer industry now has some 35 years of experience, the process of software development is still slow, difficult, costly, and error-prone. The process of DSS development is no exception to this and perhaps it is even more difficult than the development of many other systems. This is due to the following reasons: (1) many problems that the DSS is intended to help solve cannot by nature be prespecified, (2) the problem itself or the user's perception and/or conception of the problem will change over time, (3) often the user does not know his/her true needs, and (4) the DSS often should support various needs of many users. That is, it should support the solving of many problems through various decision-making styles and in many different problem situations.

The need to find more productive ways of software development in general and application software development in particular is discussed in[2] in some detail. We can identify four basic approaches that yield higher productivity in the process of software development. These are: structured design and programming, higher-level languages and special tools, the use of prefabricated pieces in construction of a new system, and automatic program generation. The complete automation of the software development process is yet a few years away, but it is certainly very desirable to move closer and closer to this ultimate goal. It seems reasonable to assume that the ultimate goal of complete automation will not be reached through one revolutionary step; rather it will happen through many evolutionary steps. Before complete automation is possible, many specialized tools must be developed to facilitate the process of software development through a semiautomatic process.

If we accept the hypothesis that many new decision support systems will be needed in the near future, it seems reasonable to focus all of our efforts on building a facility that will enable us to develop such systems with great efficiency, rather than on building individual systems in the traditional ways. This is the direction that we would like to follow.

This paper discusses the design of an environment for the development of decision support systems. We call this environment the decision support system development system or DSSDS. The system we propose will be a semiautomatic system within which a collection of highly specialized tools will be used to manufacture the individual components of a DSS from prefabricated pieces, from scratch, or from a combination of these two techniques. The individual components then could be assembled to create an integrated system. Moreover, the system would be capable of supporting the product (i.e., the developed DSS) throughout its entire life cycle.

DSS DEVELOPMENT PROCESS

Development of a decision support system requires all phases of a systems' life cycle; however, the iterations between various phases of the life cycle happen at a much faster rate. In other words, since the problem space for a DSS is continually changing, modifications and extensions of a DSS should be regarded as a norm rather than as an exception. This certainly imposes a serious constraint on the development process of a DSS. To deal with this problem we propose the following characteristics as essential features of any DSS development system (DSSDS) that is intended for the production of a successful decision support system.

1. The DSSDS should support quick production of decision support systems.
2. The decision support systems should be produced with inherent features of modifiability as well as extensibility.
3. The DSSDS should support rapid modification and production of extensions to the DSS.

The imposition of the first requirement on the DSSDS stems from a more profound reason than just the productivity gains. Since the problem itself and the user's conception and/or perception of the problem are continually changing, the DSS should be produced rather quickly, otherwise it will be obso-

lete as soon as the development process is finished. The second requirement simply states that the DSS should be built in such a way that it can be expanded (ideally indefinitely). The third requirement states that the rate of implementation of modifications and extensions should be faster than the rate of generation of needs for new modification, otherwise the system will never keep up with needs of its users and becomes obsolete very soon.

It is obvious that these requirements cannot be satisfied through traditional system development processes. There is a serious need for a more powerful facility to help satisfy these requirements. The design of such a facility will be discussed in later sections, but before that we need to talk about another very important issue in the system development process, which is prototyping.

PROTOTYPING

The key to the development of a successful system is the correct understanding of the problem by the developer. The understanding of the problem takes place in the analysis phase of the system's life cycle. At the end of this phase a formal specification of requirements is written by the analyst, which must then be reviewed and approved by the client before the design can begin. The importance of the requirements specification stems from the fact that experience has shown that errors in requirements specification are usually the last to be detected and the most costly to correct.[3,4] The importance of this stage in the system's life cycle has been well understood from the early years in the field of systems analysis and design. To overcome the problems in requirements specification, various methods and tools have been developed to assist the developer in this stage of the development. The system specification tools such as problem statement language (PSL)[5] and requirements specification language (RSL)[6] will help the analyst in checking the consistency and clarity of the specification.

The main problem with these techniques is that they rely heavily on the user to verify the accuracy and completeness of the problem specification by looking at the formal specification of the requirements. It is often difficult for the users to visualize what they see on paper as solving their problems. Besides, many users, especially the DSS users, do not have a clear understanding of their true needs prior to actual use of the system.

A second group of tools, developed with the realization of the potential difficulty of the users in verifying a printed specification of the requirements, provides graphical means to overcome this problem. Among these tools are structured analysis and design technique (SADT)[7] and SAMM.[8] Graphical representations are usually better understood by the user, provide a better picture of the system the way it has been understood by the developer, and enhance productive feedbacks. However, the user never becomes certain whether a system will satisfy his/her true needs until he/she actually starts using it.

The understanding of the true needs of the user by the developer and the user himherself can be greatly enhanced through the development of a prototype of the proposed system. Using a prototype the user can more accurately examine whether the right problem is being solved and also if he she has been understood correctly by the developer. That is, the answer to the two vital questions of the success of the system are provided with the most accuracy possible. The user, by exercising the prototype, can provide vital feedbacks to the developer. These feedbacks can be used by the developer to finalize the requirements specification. By developing a prototype the developer also will experience the difficulties and potential problems in the development process.

Thus it is clear that a prototype is a valuable learning vehicle both to the user and to the developer. In practice, however, prototypes are not built very often because of their high cost of development and also because of the additional time required for their development.

These problems of prototyping could be overcome through the use of a set of powerful tools that facilitate a relatively cheap and speedy development of a prototype.*

The DSS development system to be discussed in the next section will, among other things, provide such tools. Note that the emphasis in prototyping should not be on producing a very efficient system; rather, the emphasis should be on rapid production of a prototype that accurately reflects the requirements of the proposed system as perceived by the developer. A word of caution concerning the development of prototypes is in order: It is often necessary to make changes to the prototype in order to observe the user's reactions to modified versions. These changes should be stopped the moment no new knowledge can be learned from modification, or the cost of modification outweighs the benefits gained from it. In any event, the temptation to carry on the development of the prototype in order to turn it into the delivered system should be strongly resisted.

Prototyping in no way conflicts with the use of other systems-analysis tools and techniques. In fact, we propose that tools should be provided to the developer to help capture pertinent information from the user. This information should be stored in an organized way in a database. Automatic checking of the data's consistency and completeness should be performed, and finally, tools should be provided to the designer so that he/she can retrieve the data pertinent to each operation both quickly and in a convenient format. The developer can use this information to build a prototype with an acceptable level of accuracy for examination by the user. The requirements specification is finalized when the user is convinced the proposed system will indeed satisfy his/her needs.

THE DSS DEVELOPMENT SYSTEM

The DSS development system (DSSDS) is an environment for the development of decision support systems. The environment consists of highly specialized tools to be used by the DSS

*A good example of an existing system for rapid prototyping is KnowledgeMan," which is available inexpensively on microcomputers. KnowledgeMan has facilities for data management, ad hoc inquiry, statistical analyses, spreadsheet analysis, customized I/O screen forms, report management, and model building. Here, we are proposing an even more powerful set of tools.

developer throughout the development process to facilitate the development of a successful system. The DSSDS will increase the productivity of the developer and help him/her to produce with a moderate cost a DSS based on the true needs of the user. The philosophy of the DSSDS is based on two very simple, but also very important concepts: the use of highly automated tools throughout the development process and the use of prefabricated pieces in the manufacturing of a whole piece whenever it is possible. The first concept increases the productivity of the developer in the same way an electric saw improves the productivity of a carpenter using a hand saw. The second concept increases the productivity of the developer analogous to the way a prefabricated wall increases the productivity of the carpenter building a house.

Although many of the tools that DSSDS provides could be used in the development of any application system, our emphasis would be towards tools that are helpful in the development of a DSS in particular. By specializing we expect to gain efficiency in the development process because there will be a real need in the future for the development of many decision support systems. Design of any large system for the first time is a major task. To insure the success of the system, it is not recommended that a gigantic system be designed from the beginning, in the expectation of supporting the development of every detail of the system. Rather, in the design of a DSSDS we follow the evolving characteristic. That is, we think that a nucleus DSSDS should first be designed and developed to support the essential needs of the developer. However, the system should be extensible so other features can be added to it when the need for them becomes apparent. Nevertheless, the DSSDS should have the following characteristics:

1. The DSSDS should support the development of a successful DSS.
2. The DSSDS should support the development process throughout the entire life cycle of the system, that is, it should support capturing of the requirements from the user, development of the prototypes, design and implementation of the delivered system, testing, and finally the maintenance of the DSS.
3. The DSSDS should support the development of different decision support systems in different programming languages and possibly for different target computers.
4. Various tools of the DSSDS should be relatively easy to use and independently available.
5. The DSSDS should be capable of evolving over time.

In this context the independence of various tools implies only the functional independency; however, coordination of various tools is essential. The evolving feature of DSSDS here means that the system should allow new tools to be added to the system as well as allow old tools to be improved or replaced by more advanced tools.

AN OVERVIEW OF THE DSSDS ENVIRONMENT

The DSSDS environment can be thought of as a workshop with many tools and prefabricated parts that the developer can use throughout the process of building a new DSS or to upgrade or repair an existing DSS. The environment of the DSSDS is shown in Figure 1. The developer is provided with a development language (DL), which is basically a powerful command language. Modules can be written in command language or in any other programming languages such as FORTRAN, COBOL, or PASCAL. By module we mean any set of executable lines of code that has a name and is written to do a certain job. A module can be used independently, or it can be used in conjunction with other modules to build a more complex module. A module can do computation, perform read and write operations, transform data, or perform any other computer operations in order to achieve a certain objective.

In addition to the development language, a number of other facilities are available to the developer. These are systems analysis and design facility (SADF), a model management language (MML), a screen management language (SML), a source code manager (SCM), a report generator (RG), a graphics generator (GG), and a request handler (RH). Each of these facilities can be used through the command language or independently.

The Development Language (DL)

The development language is a command language. Its function is to provide a host to other facilities, as well as to provide a collection of useful functions to be used by the developer. Individual commands or procedures written in MML, RG, GG, and so on can be invoked from the DL. The command language provides interface between modules written in various facility languages (i.e., MML, RG, GG, etc.) as well as modules written in programming languages like FORTRAN, COBOL, PASCAL, and so on. In this way the developer can write a program whose components are written in different programming languages and/or use various development facilities. For example, to create a plot of the predicted sale for years YR1 to YR2, the following program can be written:

```
RETRIEVE (SALE, YR, R10)
CALL REGRESS (SALE, YR, COEF)
CR FYR (10) = YR1 TO YR2
CALL FORCAST (COEF, FSALE, FYR)
GG. PLOT (FSALE, FYR)
```

The first line is intended to retrieve the sale values with the corresponding year values (YR), for ten most recent years (R10). The second line will run a regression on sale as a dependent variable against YR. Then, variable FYR is defined to be an array with values YR1 to YR2. The fourth line runs a forecasting model using the coefficients produced in line two. Finally, line five invokes the command PLOT from a graphics generator (GG) to plot the predicted sale against the future years.

By being able to create a program whose components are written in different languages, we benefit in two ways: First, each component can exist in its most efficient form. That is,

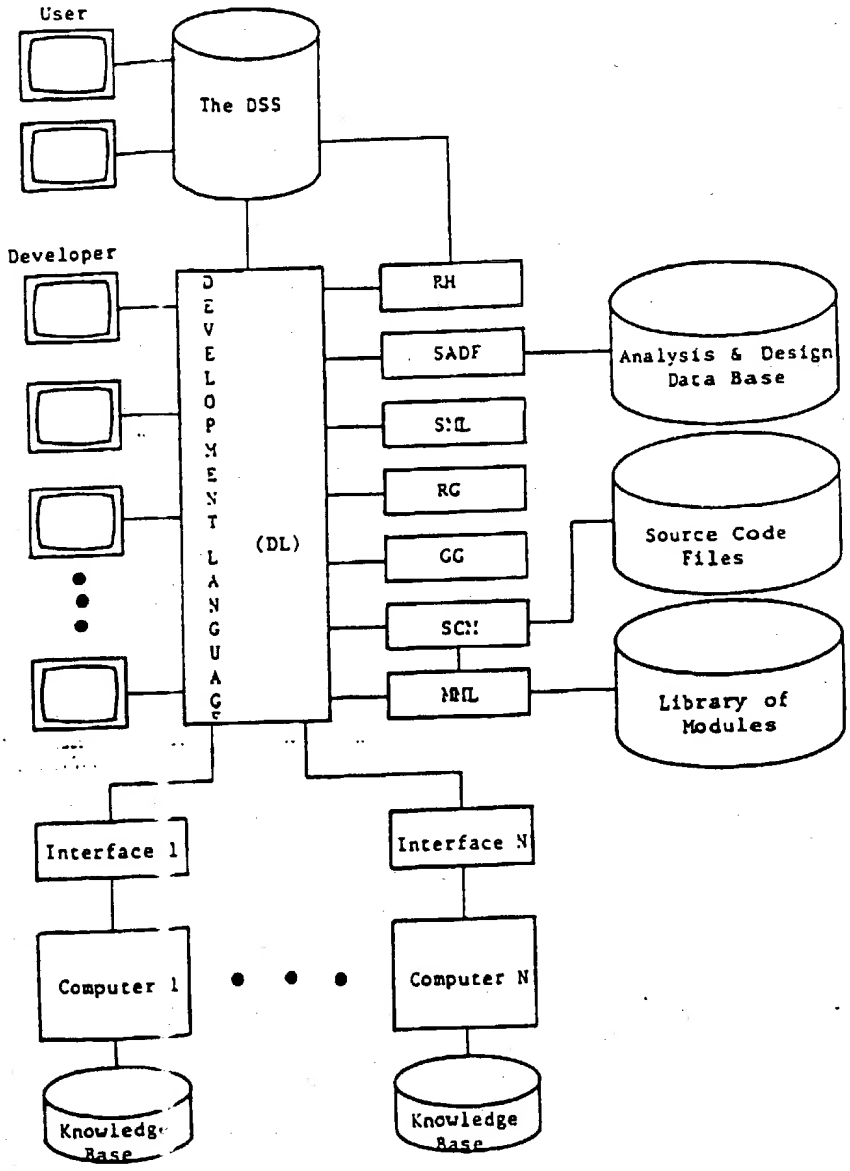


Figure 1—An overview of the DSS development system (DSSDS)

each module can be written in a language that is most suitable to its function. Second, more productivity can be gained by using many existing modules currently available in different programming languages. The purpose of this paper is not to discuss the syntax or semantics of the command language; rather it is to present the concept of such a language. A sample of some other commands is shown in Figure 2.

Systems Analysis and Design Facility (SADF)

Development of a DSS, like any other system, starts with analysis. The aim of the systems analysis phase is to gather enough information about the needs and operations of the system so that any qualified data-processing professional will by reviewing this information be able to understand what the needs and requirements of the new system are and what it is supposed to do. The purpose of the systems analysis and design facility (SADF) is to help solicit pertinent information from the user, to store and organize this information in a database, and to check the consistency of the information and make it available to the developer in a usable form.

Development of any nontrivial information system generally requires the participation of many people. One problem facing the development of such systems is the documenting of

CREATE x	Create a file and call it x; if x is not present a working file is created.
STORE x	Store file x. The system will prompt for the location of the file and security feature. The default for x is the current working file.
SAVE	The system will save the entire work of the session as it is, so it can be continued at a later time.
RECREATE	The system will recreate the working environment as it was left off in the last session.
EXECUTE x, (C = Comp, I = data, O = y)	The system will send module x to computer comp to be executed using file "data" as input file, and sending the output to file y. Defaults are the main frame computer and terminal input/output respectively.
EXTRACT x, y, type	Extract file x and place it on file y. Type can assume values M or D for module and data. If system is unable to find the location of x, prompts for help.
RETRIEVE (x,y,z,...,pi)	Retrieve i instances of variables x, y, z, etc. P = R (recent), F (first), or A (all).
CR x(m) = i,j,k...	Create a vector of length m and initialize its values to i, j, k, etc.
CR x(m) = i to j	Create a vector of length m and initialize its elements to values from i to j.
CR x(m,n) = i_{11},i_{12}, ...,i_{mn}	Create a table and initialize its values row by row to $i_{11},\ldots,i_{mn}$. If no values are given the table is created but is not initialized.
IF (exp) command	Conditional execution of a command.
DO WHILE (exp) . . . end	Looping while "exp" is true.
DO FOR i = j,k . . . end	Looping

Figure 2—A sample of features of the development language

the important communications among these participants so that at each point in time it is clear what decisions have been made in the handling of each component of the system. SADF will store these communications in a network database and will relate them to the originator of the comment as well as to the component about which the comment is issued (see Figure 3.) This information is available to all participants in the development process and can be accessed by simple commands or queries.

In the course of system development, some of the necessary information for the formulation of system requirements can be captured from existing systems or existing documents, but the ultimate source of the information is the user. When developing a DSS, it is very unlikely, because of the newness of the field, that an old computer-based DSS will be in place before the development of a new one. Therefore, the user remains the only reliable source of information. However, different users have different needs and viewpoints that some-

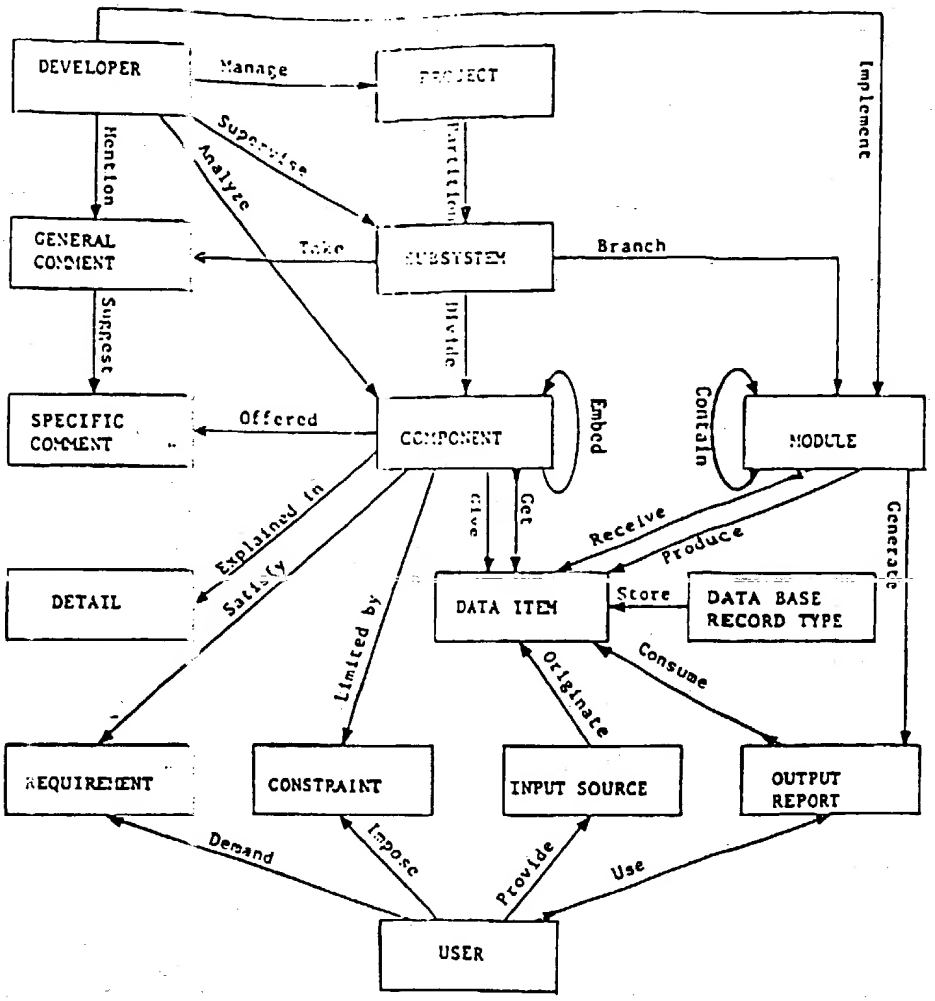


Figure 3—Extended network logical structure of the SADF database

times are in conflict. In any case, all viewpoints should be heard and all reasonable needs should be accounted for, according to some priority list. SADF stores this information in an extended network database[10] along with other information pertaining to the analysis and design of the DSS.

Part of the requirements could be obtained from the user through a program that would interview the user in a conversational mode through an interactive terminal. This could be easily accomplished by a questionnaire designed especially for solicitation of information from the user; however, instead of a human interviewer, the computer can be programmed to conduct the interview through an interactive terminal. Questions will be presented to the user, and answers will be obtained and stored in a database. After interviewing all users, a summary report will be produced and the results stored internally so that the report can be viewed by the analysts, designers, programmers, and so on. An automated interview usually is not sufficient to capture all requirements; however, it can help the analyst by revealing the problem areas requiring more extensive study. In any event, all obtained information will be stored in an extended network database (Figure 3). This method of storing the information facilitates the effective use of the information and provides an excellent means of documentation. A detailed discussion of SADF appears in Reference 2.

MODULE MANAGEMENT

One way to achieve high productivity in the process of software development is to use prefabricated pieces in the construction of a new system. The use of preprogrammed modules in the manufacturing of a new system not only increases the productivity of the software development process, but also increases the opportunity for producing high-quality software. Production of higher-quality software is possible in two ways:

First, the frequently used modules can be fine tuned to perform very efficiently. That is, these modules can be written in assembly language or they can be written by highly skilled programmers. Second, since preprogrammed modules presumably have been in use in other systems and environments they have been perfected. Also, the performance of these modules has been observed in actual practice, so their strengths and weaknesses are better known. The developer is therefore building his/her system with a better-known material so it is expected that a better system will be produced. In practice, however, the use of prefabricated pieces in the development of a new system is negligible, unless the same person is developing a similar system. The main reasons for not using the product of previous efforts in the development of a new system can be classified in the following categories:

1. Inflexible design—The module does not directly fit the current need, and inflexible design does not permit easy modification of the module.
2. Different programming language—The module is written in a different programming language with no interface to the language used for the system development.
3. Machine dependence—The module is written for a particular machine and cannot be used on other machines.
4. No organized information about the existence of the module exists—The modules are scattered in various places (e.g., files, tapes, computer cards, etc.). No one knows about their existence or there is no convenient way of getting information about them.
5. Lack of documentation—The existence of the module is known, but there is lack of documentation. The author is either unknown or is no longer with the organization, therefore, no one is sure how to use the module.
6. Lack of information about reliability of the module—The developer simply cannot trust someone else's product without having some evidence about the reliability of the product.
7. Lack of performance data about the module—There is no evidence to indicate how the module performs in practice.

If we are able to find a solution to these problems then we can expect to produce quality software with high efficiency and with reasonable cost.

The first problem calls for flexible design. Flexible design under the DSSDS is possible. Modification of a module through a source code manager (SCM) is also greatly facilitated. The second problem is solved under the DSSDS development language (DL) because DL provides interface to several programming languages, and any program written in DL can call modules of different programming languages. The third problem is less severe because most of the programs written in high-level languages are portable. There are several ways that this problem can be solved. If there is a complete program it can be routed to the right machine to be executed and the results transmitted to the originator of the problem. If the number of machine-dependent modules is considerable, a virtual machine (a simulation of another machine on an existing machine) can be developed to run these modules. Translators can also be written to translate programs of one given machine to another. The first solution is supported by DSSDS. The others can be designed and added to DSSDS if economically justifiable.

In this section we present a solution to problems 4 through 7. To solve these problems we need to create a centralized information base that contains all necessary information about all modules available to the software development center. This centralized information base can be used by individual members of the development team to select the appropriate modules to be incorporated in the development of the new systems.

To centralize all pertinent information about various modules, we design an extended network[10] database system, which we call the library of modules (LOM). We show how this library can be used to assist the developer in the task of module selection as well as to provide him/her with informative information in each problem area.

Our interest in preprogrammed modules is not stimulated only by productivity gains and production of quality software, but also because in the development of a DSS we need to supply the DSS with a collection of modules to be used by the problem processing system (PPS) and/or decision maker for model building activities. Therefore, we consider the library of modules an essential part of our DSSDS.

The Library of Modules (LOM)

The library of modules stores all desirable information about available modules in a centralized fashion. The developer after designing his/her system can turn to the module library and see which of the existing modules can be used in the development of the new system. If none of the modules can be used directly, the developer may then investigate if any of the modules can be used with minor modification. If the module library is large enough, it is reasonable to assume that some modules will be useful in the development of a new system. This will help a speedy development of a new system, which we consider an essential requirement for the development of decision support systems. This approach also provides an opportunity for developing high-quality software with reasonable cost.

The logical structure of an extended network database along with a proposed list of data items is shown in Figure 4. Modules are categorized by the problems they solve and the problems themselves are categorized by subject area. There may be more than one module for solving a given problem and a given module may solve more than one problem (N:M relationship). For each module, information about the name of the module, module number (similar to the call number for books), the purpose (what does it do?), technique used, and origin (where did it come from?) is stored. The record type THEORY is intended to represent the scientific basis of the technique used in the development of the module. The developer can check to see if there is a sound scientific basis for the technique, and if so, can educate him/herself and learn about the conditions under which the technique is valid. In other words THEORY does the job of a handbook. This can be very

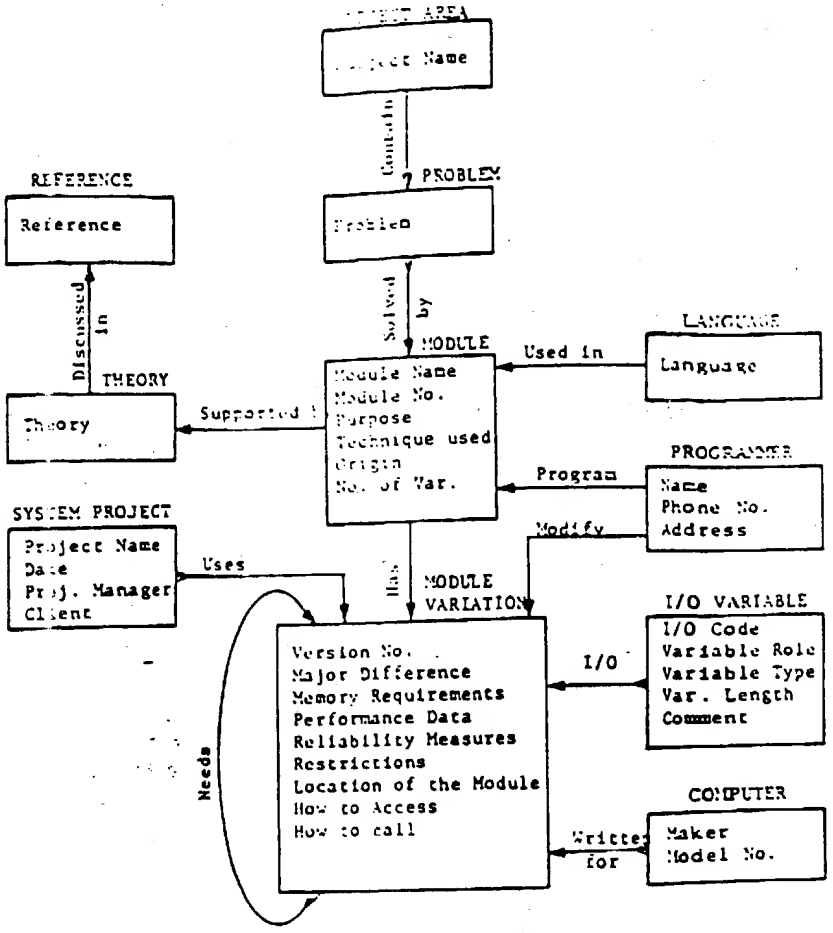


Figure 4—An extended network structure for the library of modules

helpful since the developer is not necessarily knowledgeable in all problem areas. For each THEORY a number of references are also given. Each module may have many variations (it is assumed that there is at least one variation, i.e., the original). Each record occurrence of the MODULE VARIATION record type contains the properties of a specific variation. These properties are shown in Figure 4. Major Difference is an explanation of the major difference between this version and original version of that module. Memory requirements gives the size of the program in bytes and is especially helpful when there is a memory restriction. Reliability and performance data essentially tell how reliable the module has been and how fast it runs. The other information includes restrictions of that variation, where it could be found, how it should be accessed, and what the calling procedure is. The record occurrence of each particular variation is associated with the system project(s) in which it has been used, and for each project the names of the project manager and client as well as the name of the project and date of development are given. So if the developer wants additional information about the development process or practical results he/she can contact the appropriate person. Each occurrence of record type LANGUAGE is related to all modules written in that particular language. So it is possible both to find out in what language a particular module is written and to scan through al modules written in a given language. Some variations of a module may be written for a particular computer so the record type COMPUTER is related to record type MODULE VARIATION through the many-to-many set, Written for. Each module variation is linked to its input/output through the set I/O. Properties of each I/O variable are stored in an occurrence of I/O VARIABLE. The I/O codes of I, O, or I correspond to input variables, output variables, and both respectively. Other data items of I/O variables are shown in Figure 4. Each module is linked to its programmer and each variation is linked to the programmer who did the modification. Each programmer's name, telephone number, and address is given so additional information can be obtained from the programmer if necessary.

Thus the library of modules (LOM), directly or indirectl

(through references, addresses, etc.) includes all the information that the developer would like to know about a particular module. Another interesting feature of LOM is the set relationship Needs, which will be discussed in the next section.

Module Dependencies

Within a system it often happens that the output of a module is used as input by another module, thereby creating a dependency between the two modules. We call this dependency between two modules a context-sensitive association or a weak dependency, because the association is the result of input/output needs rather than the result of the direct need of one module for another. The dependency is context sensitive because it very much depends on the context; if module *x* in a given system needs the output of module *y* in order to work, in a different context (i.e., a different system), this may not be the case, because the input of *x* may be provided in another way (e.g., be simply read in).

In contrast to this dependency there is another kind of dependency, which we call strong dependency or a context free association. A strong dependency is the result of one module calling or invoking another module. For example if module *z* calls for the service of module *y* in its procedure, then we say *z* has a strong dependency on *y*, because *z* cannot function unless *y* is present. *y* in turn may have strong dependency on another model. In Figure 5, module *z* has strong

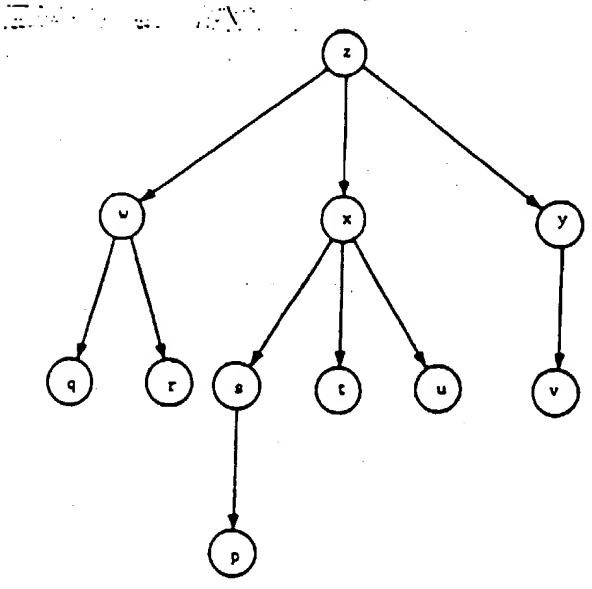


Figure 5—Strong dependencies among modules

dependency on *w*, *x*, and *y*; *w* in turn has a strong dependency on *q* and *r*; *x* is strongly dependent on *s*, *t*, and *u*; *s* in turn has strong dependency on *p*; and finally *y* is strongly dependent on *v*. These dependencies are context free because no matter in which system we use model *z*, it needs modules *w*, *x*, and *y* in order to operate. Modules *w*, *x*, and *y* in turn need the service of their own modules. This hierarchy continues until all the new modules stand alone and are self sufficient.

This strong dependency of one module on other modules is effectively captured by the recursive relation, Needs. This is an *N:M* relationship because each module may need the service of several other modules, and each module may give service to many modules. Treatment of module dependencies in this way greatly facilitates the development of new systems as well as modeling activities. Notice that if module *z* is selected to be included in a new system, all the modules that *z* is dependent on for service in a direct or indirect way should go with module *z*.

The linkage through set Needs provides valuable information to the developer. For example, if a module like *z* is a candidate for selection, the developer can scan through all other modules that are directly or indirectly needed by *z* and examine such properties as performance data, reliability measures, the language they are written in, hardware dependencies (if any), and so on. Examination of this information is important because it may reveal some undesirable properties of one or more modules in the collection, which may require the rewriting of those modules or the selection of an alternative module.

Another valuable benefit of this approach is that since through this linkage the developer can find out which modules use the service of a given module in a direct or indirect way, it is very easy to find out which modules will be affected by alteration of the given module, and therefore appropriate measures can be taken if necessary.

A third advantage of this approach is that it eliminates redundancy in the storage of modules. In other words, each module is stored only once, no matter how many other modules use its service.

Other consequences of this approach are that the system can evolve and can become personalized. The evolution is possible because new independent or dependent modules can be added to the system without difficulty. The developer can also use the original primitive modules and can build upon those a collection of modules to be used by him/herself on a personalized basis.

The system can also display a learning behavior. Observe that the set of modules that are directly needed by a module such as *z* can be considered as preconditions to *z*, because without them *z* cannot be executed. However, existence of a module in the database of the LOM automatically means that the preconditions are satisfiable, and in fact the linkage paths represent the solution paths to satisfy the preconditions. Any time a new problem is solved, that is, a new module is formulated with or without the use of existing modules, this new information is added to the system and the problem need not be solved again because the solution path to this problem already exists in the database. Thus the system displays a learning behavior. Moreover, these new skills are acquired in the area for which the system has been used and for which they are presumably needed the most. In other words, the system learns the right things. A final comment on the learning feature is in order: If the original collection of the primitive modules is considerably large, chances are that most of

the new modules can be created through the use of the existing modules. It is the job of a human or computerized problem solver to combine the right ingredients to create a module that can deliver the desired results for a given task. It is expected that most new modules will result from combining the existing modules or from using some parts from the existing modules rather than from being created completely from scratch. Different schemes should result in different environments best suited to different lines of development.

Extensions to the Library of Modules

By making some conventions we can also add the information about the weak dependencies to the database. A weak dependency is the result of one module using the output of another module as its input. But inputs to a module generally can be provided by a variety of sources. For example, more than one module can provide input that can be used by a particular module. The inputs can also be read from a database, file, cards, and so on. So there are alternatives for the developer to choose from. The approach preferred depends on the kind of raw data available at a given context. It is beneficial to the developer if he/she is reminded of his/her choices. To include this new information we do not need to change the structure of our database but we need to make a few conventions. First we distinguish between three kinds of modules: a process module, which is a regular module and performs some data-processing task; an input module, which provides the inputs to a given module by reading them from the tape, from the database, from cards, and so on; and a link module, which links a process module to its alternative input modules. We let all three types of modules share the same record type; however each occurrence contains the information about the type of that module.

To help clarify this problem, let us consider an example. In Figure 6 module *z* needs modules *x and y and* some input that can be provided in three different ways. Either it can be provided by module *I1* by directly reading from some input source (e.g., from the database), *or* by module *I2* by directly reading from a different input source (e.g., from cards), *or* it can be provided as an output of module *w*. Module *w*, in order to work, needs module *v and* some input that can be provided in two alternative ways of *I3 or I4*. Notice that the *I* modules represent input modules and they are always terminal nodes in the dependency tree. The *L* modules are link modules and they always branch into alternative modules that can provide the input to the so-called owner module. Only one of the alternatives is necessary and sufficient to provide the input. The ordinary modules like *x*, *y*, and *w*, can call any of the other two types of modules or be self-sufficient.

Here the developer is provided with different alternatives for solving the problem although he/she may use the same module *z*. He/she may prefer one alternative over others in a given context or he/she may include some or all of the alternative solutions in the new system he/she builds and then let the user decide about a convenient approach in each problem situation.

Observe that Figure 6 closely resembles an AND/OR

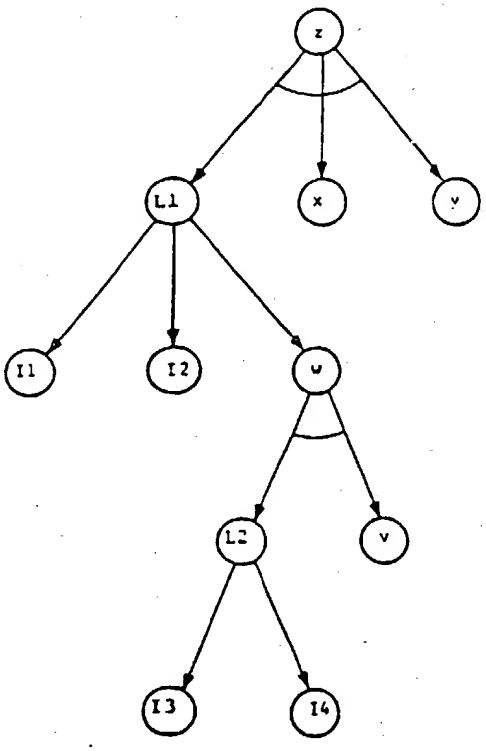
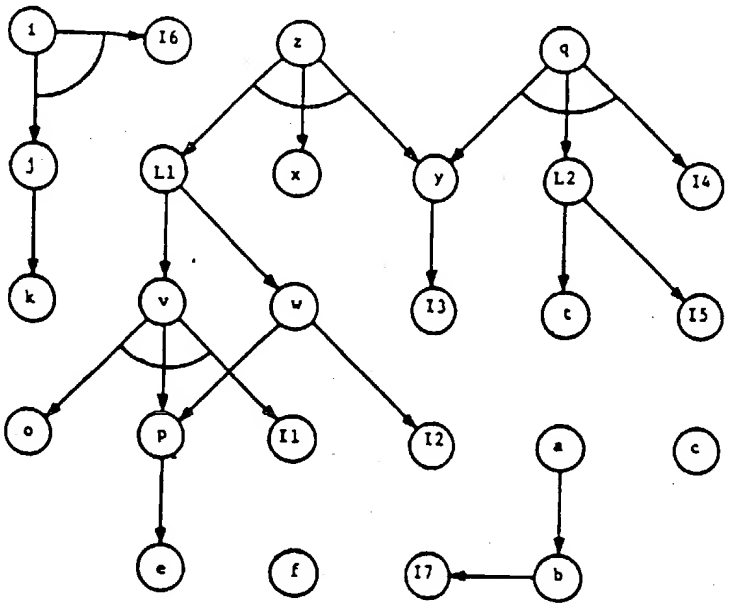


Figure 6—Strong and weak dependencies among modules

graph.[11,12] The process modules if they branch, represent AND or synthesis nodes, and the link modules represent the OR nodes. Since AND/OR graphs are used in the problem reduction approach to automated problem solving,[13] it follows that our database technique could be used as an effective mechanism in automatic problem solving. The complete AND/OR graph can be represented by the relationship Needs in the LOM database. Each node contains the information about whether it is an AND node or an OR node, and each linkage represents a reduction operator. Notice that the problem solving in this way is reduced to a search through the database. Moreover, if the start symbol (i.e., the module we are looking for) is found directly in the database, then the solution is guaranteed, provided the input data can be prepared in the right form. The system offers flexibility by allowing the input data to be fed to the module in various forms depending on the context. Different inputs may result in different combinations of modules that deliver the same results.

Alternatively, our module linkage mechanism can be the representation of a production system.[13] In other words, this mechanism can be used as a storage mechanism for a production system database (PSDB). If we consider our database as a representation of a production system, then all dependent modules are considered as nonterminals and the independent modules (I/O modules and self-sufficient modules) are considered as terminal nodes. Figure 7 shows the relationships of modules and their corresponding production system. Note: the collection of linkages emanating from a process module



1) i → j, I6
2) j → k
3) z → L1 | x, y
4) L1 → v | w
5) v → o, p, I1
6) v → p | I2
7) y → I3
8) q → y | L2 | I4
9) L2 → t | I5
10) a → b
11) b → I7
12) p → e

k, I6, o, e, f, I1, x, I7, I2, I3, t, I4, I5, and c are terminal nodes (i.e., stand alone modules).

Figure 7—Production system for modules in the library of modules

represents one production while each linkage emanating from a link module represents one production.

Module Management Language (MML)

The library of modules contains the information about any module accessible through the development environment. The source code of these modules may be stored in source-code files under the source code manager (SCM), or it may be stored in other files even under other computer systems. Regardless of the location of the module, all the information about its properties, location, and the procedure for accessing it is stored in the LOM. To use this information the developer needs a collection of tools so he/she can easily scan through the information in the library and select the desired modules. After the selection of the modules the developer wants to copy the module itself plus its supporting modules to an appropriate place to be included in the new system with or without some modifications.

The use of a network database management system for the library of modules automatically provides the developer with a powerful tool for retrieval and manipulation of information in the LOM. That is, the user can use the query language of DBMS and question the informational content of the database and/or manipulate the data. The developer can also develop a set of macrocommands that he/she can use repeatedly. Nevertheless, the existence of a module management language (MML) greatly facilitates the job of the developer. A set of basic commands is shown in Figure 8. Additional commands in the form of macros can be designed by the developer on a personalized basis and be added to the system. The MML is intended to be conversational in the sense that any ambiguities may be resolved through conversation with the user.

OTHER DEVELOPMENT FACILITIES

In the design of the foundation of DSSDS we implicitly assumed that a database management system (DBMS) exists. Moreover, we based our design on a network database system. Although it is possible to design a DSSDS without a database management system, existence of a DBMS greatly facilitates the design and implementation process. Besides,

ADD<rt>	to add a new occurrence of record type "rt" in the data base
DELETE<rt>	to delete an occurrence of record type "rt" from the data base
CHANGE<rt>	to change data item(s) within record type "rt". (The system prompts for additional information.)
DISPLAY<rt><x>	to display the informational content of occurrence x of "rt"
DISPLAY<rt>.<y>.<st>	to display the informational content of record type "rt" for all members (or owners) of owner (or member) y of set st
DISPLAY<rt>	to display all occurrences of rt. ("SYSTEM" is assumed to be the owner, otherwise, system prompts for the owner.)
DISPLAY OWNER<rt><x>.<st>	to display owner(s) of occurrence x of record type rt through set st
DISPLAY MEMBER<rt><x>.<st>	to display all members of occurrence x of record type rt through set st
DISPLAY SUBMODULE<x>.<it>	to display the value of item type "it" for all submodules directly needed by x, if "it" is missing all items will be displayed
DISPLAY ALL SUBMODULES<x>.<it>	to display the values of item type "it" for all direct or indirect submodules of x
DISPLAY SUPER MODULE<x>	to display modules that directly use the service of module x
DISPLAY ALL SUPER MODULES <x>	to display all modules that directly or indirectly use the service of module x
COPY<x>,<y>	to copy module x to file y
COPY ALL<x>,<y>	to copy x and all modules needed by x (directly or indirectly) to file y

Figure 8—A set of commands for a module management language

since the DSSDS normally would be used in a development center, existence of a DBMS in such a center is unquestionable. We also assumed the existence of a query language that would work with the database system.

In Figure 1 the existence of a report generator (RG), a graphics generator (GG), and a screen management language is recognized. We do not intend to discuss these facilities because these facilities do exist in a variety of forms. The report generator and graphics generator that we have in mind should have features similar to those of NOMAD,[14] for a screen management language we would like to have display facilities similar to those of SPF[15] or SCREEN MASTER.[16] The source code manager (SCM) is a tool that facilitates the generation of new modules or the alteration of existing modules. A detailed discussion of the SCM appears in Reference 2.

Request Handler (RH)

The request handler (RH) is intended to be used for maintenance purposes while the DSS is in operation. The purpose of the RH is to provide a communication link between the DSS and the DSSDS. The RH performs several important functions. First, suppose while the DSS is in operation, a bug is found in one of the modules. The user then sends a request through RH explaining the problem. The user does not necessarily know which programmer was involved in the development of that module. The RH by looking at the LOM can route the message to the right programmer. In case the programmer is unknown or is no longer with the organization, the RH will route the problem to the person in charge or the least-busiest person in charge of such problems.

Second, suppose that the user wants some extensions. That is, the user needs a new model that is not found in the DSS and that cannot be formulated through existing modules in the DSS by PPS or by the user him/herself. The RH will look at the LOM; if the module is found in the LOM, the RH will automatically access the module and route it to the DSS. Otherwise, the RH will place a message in the mail box of the least busiest developer or the developer with the right qualifications for that job. In case a user of the DSS has some questions and needs some help, he/she can send a help request to the RH. The request handler starts a dialogue with the user and gathers information about the subject and the nature of the question and then routes the message to an appropriate developer.

Through the RH the communication link between the DSSDS and the DSS remains open throughout the system's life cycle. Through this link the news about the availability of new modules, new versions of the existing modules, or new facilities can be sent to the DSS to be placed in the mail box of interested parties. RH provides a valuable facility for supporting the product during the operation phase of its life cycle.

DSS DEVELOPMENT

The DSSDS satisfies all the requirements we stated earlier for a DSS development system. That is, the DSSDS supports a speedy development of a DSS and it also supports the DSS in its entire life cycle. Various decision support systems can be developed through the DSSDS for different needs. The tools of the DSSDS are available independently, and finally, the DSSDS is capable of evolving over time.

With the initiation of a DSS project, systems analysis begins. SADF helps the developer to gather the information and store it in an organized way in a SADF database. Through a SADF all members of the development team can use the same data and share their thoughts. When the developer believes he/she understands the problem correctly, the development of the prototype begins. In prototyping the emphasis is on speedy development of a system that reasonably represents the proposed system. Through DSSDS a speedy development of a prototype is possible because the LOM can provide considerable preprogrammed modules. Besides, the modification of existing modules is greatly facilitated under the SCM. The Report Generator, the Graphics Generator and the query language are excellent facilities for prototyping, because efficiency is not an immediate concern in prototyping. For example if a special report has to be prepared, it is very likely that the report could be provided through RG quite easily. However if and when the report proved to be necessary and needed on a recurring basis then a new module should be written to create this report very efficiently for the final system.

SUMMARY

The need for many decision support systems in the near future stimulated our interest in finding a convenient way for developing such systems. The changing nature of DSS required us to find a way for speedy development and fast modification of DSS. Our study resulted in a proposal for a DSS development system (DSSDS). The DSSDS facilitates both the development and maintenance of a DSS. The philosophy of the DSSDS is based on two concepts: the use of highly automated tools throughout the development process and the use of prefabricated pieces in the manufacturing of a whole piece. The environment of the DSSDS consists of a development language (DL), a systems analysis and design facility (SADF), a module management language (MML), a source code manager (SCM), a report generator (RG), a graphics generator (GG), and a request handler (RH).

The DSSDS provides an environment in which the developer can create high-quality decision support systems, with moderate cost and in a relatively short period of time.

REFERENCES

1. Martin, J. *Applicanon Development Without Programmers*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.
2. Ghiaseddin, N.. "Framework for a DSS Development System," Ph.D. dissertation, Purdue University, 1982.
3. Boehm, R., "Software Engineering—R and D Trend and Defense Need." In *Research Directions in Software Technology*. Cambridge, Mass.: MIT Press, 1978.
4. Gomaa, H., and D. Scott. "Prototyping as a Tool in the Specification of User Requirements." *Proceedings of 5th International Conference on Software Engineering*, March 1981.
5. Teichrow, D., and E. Hershey. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering*, January 1977.
6. Bell, T., D. Bixler, and M. Dyer. "An Extendable Approach to Computer Aided Software Requirements Engineering." *IEEE Transacnons on Software Engineering*, January 1977.
7. Ross, D., and W. Schomaman. "Structured Analysis for Requirements Definition." *IEEE Transacnons on Software Engineering*, January 1977.
8. Stephens, S., and L. Tripp, "Requirements Expression and Verification Aid." *Proceedings of the 3rd International Conference on Software Engineering*, May 1978.
9. *Knowledge Manager Reference Manual*, Micro Data Base Systems, Inc., Lafayette, Indiana, 1983.
10. MDBS INC. *MDBS Application Programming Reference Manual*. Lafayette, Ind., 1981.
11. Nilsson, N., *Principles of Artificial Intelligence*. Palo Alto: Cal.: Tioga Publishing, 1980.
12. Bonczek, R., C. Holsapple, and A. Whinston, *Foundation of Decision Support Systems*. New York: Academic Press, 1981.
13. Davis, R., and J. King. "An Overview of Production Systems." In E. Elcock and O. Michie (eds.), *Machine Intelligence 8*. New York: Halsted Press, 1977.
14. McCracken, D., *A Guide to NOMAD for Application Development*. Wilton, Conn.: National CSS, 1980.
15. Joslin, P., "System Productivity Facility," *IBM System Journal*, 20 (1981).
16. *SCREEN MASTER Reference Manual*, Micro Data Base Systems, Inc., Lafayette, Indiana, 1982

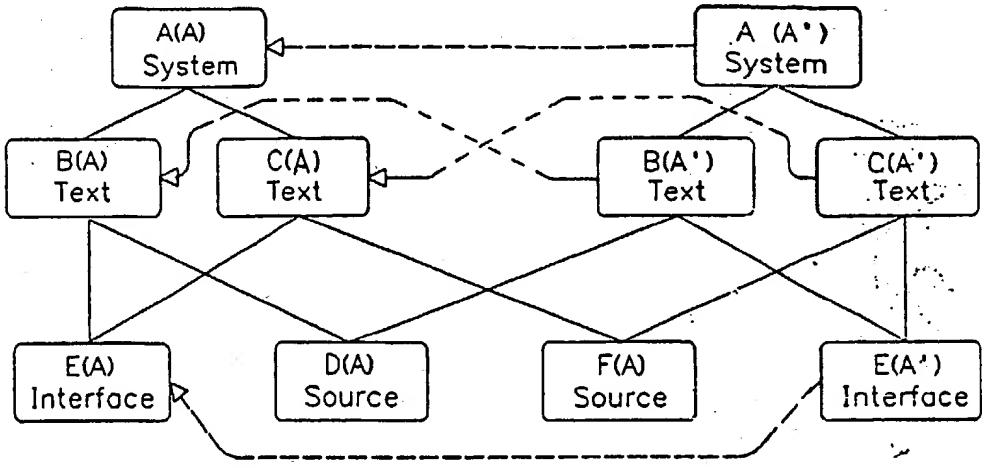
700 IBM Technical Disclosure Bulletin
30 (1987)Oct., No.5, Armonk,NY,USA

IBM Technical Disclosure Bulletin Vol. 30 No. 5 October 1987

p353-355

G06F9/44G

METHOD FOR AUTOMATED ASSEMBLY OF SOFTWARE VERSIONS



A new version, or configuration, of a software system is automatically derived from a predecessor configuration as constituent modules of the predecessor are updated.

The structure of a software system may be represented as a directed, acyclic graph with a single root node. The graph provides directions for the assembly of the system. Assembly begins at leaf nodes and proceeds toward the root node. An example of such a graph is shown in the accompanying figure.

In the figure, a plurality of system modules are shown. The solid lines extending therebetween correspond to structural links which indicate the structural dependency of modules. In particular, the system module A(A) is derived from two text modules B(A) and C(A). Module B(A) is derived from a source code module D(A) and an interface module E(A). Module C(A) is derived from modules E(A) and F(A).

If a module is altered, the modules dependent thereon are automatically changed to account for the updating. This is illustrated in the figure where a change in the interface module E(A) is depicted and represented (and renamed) as module E(A'). Because module E(A) was used in generating modules B(A) and C(A) which, in turn, were used in deriving system module A(A), these modules are automatically replaced by updated (renamed) modules B(A'), C(A'), and A(A'), respectively. The replacement aspect is suggested by the dashed line representation. The figure thereby depicts the result on the system of modifying module E(A).

METHOD FOR AUTOMATED ASSEMBLY OF SOFTWARE VERSIONS - Continued

To aid in determining when a new version of a module must be created, each module is tagged with the module id of the root node in the graph in which the module was originally created. For example, B(A) indicates that module B was originally created in the system whose root node is A.

A general procedure for creating a new version of a module is set forth below in two phases:

1. An upward scan is made in the system structure graph of all nodes dependent on the affected node, and a work list is created consisting of nodes whose tags are not the same as the identifier of the system structure graph. This list must be ordered by the maximum distance of the nodes from the affected node for the next phase to work properly. The affected node must be the first node on this list, as its distance is zero. The nodes on this list will have been inherited from a predecessor system version, and will be affected either directly by the modification, or indirectly by being constructed differently (having different dependents). New nodes must be created in the current system version graph for each node on the work list, and this will be performed in the next phase of the method. In phase 1:

   a. initially place the affected node of the work list;

   b. set the current node pointer to the first node on the work list;

   c. if the current node has a system version tag which is the same as the current system version graph, skip to step 1f;

   d. select without replacement an ancestor of current node that is in the current system version graph;

   e. if the ancestor is already in the work list, move it from its current position in the list to the end of the list; otherwise, add it to the end of the work list;

   f. if another ancestor exists, skip to step 1d;

   g. move the current node pointer to the next node in the work list;

   h. if the current node pointer points to a node, then skip to step 1c; else, proceed to phase 2.

2. For each node on the work list, in the order of the work list, the following steps are carried out:

METHOD FOR AUTOMATED ASSEMBLY OF SOFTWARE VERSIONS - Continued

a. if the current node's system version tag is the same as the current system version graph name, then skip to step 2e;

b. create a new node;

c. assign the new node a system version tag that is the name of the current system version graph;

d. link the new node to its predecessor, the old node;

e. for each immediate dependent of the old, inherited node, if the dependent has a successor tagged with the identifier of the new system version graph, then create a construction link from the new node to the successor of the dependent of the old node; otherwise, create a construction link from the new node to the dependent of the old node.

In addition to providing a process of updating system description files automatically, the above algorithm permits multiple programmers to work simultaneously on revising a module in a current version. Each programmer can automatically generate a respective "mini-version" which may be compared with the mini-versions of other programmers. If there is no overlap in changes, the mini-versions can be straightforwardly combined; otherwise, the programmers merge the mini-versions together.

EUROPEAN PATENT OFFICE

SOURCE: (C) IBM CORP 1993

XP-002083221

pp. 02-1990 3

AN : NA9002141

PUB : IBM Technical Disclosure Bulletin, February 1990, US

VOL : 32

NR : 9A

PG : 141 - 143

TI : Robotic Wafer Handling System for Class 10 Environments.

TXT : - - A technique is described whereby a robotic wafer handling system, as used in the fabrication of integrated circuits, provides automatic cycling of wafers in a class 10 clean room environment, by controlling particle producing mechanisms.

- - Referring to the drawing, the system is a self-contained unit consisting of base 10, robot unit 11, multi-gripper 12, four cassette nests 13, aligner 14, carrier spreader 15, idle stand 16, X, Y, Z-sensing device 17 and light curtain 18. Mounted in base 10 are the controls, consisting of a central processing unit, to drive the robot. An interface circuit card (not shown) provides the electrical connection to an external vacuum processing system for the control of pneumatic valves for robot unit 11. Numerous strategically placed sensors interconnect to the central processing unit, located remotely from the system, for controlling the mechanism's movement while in operation.

- - Top mounting plate 19 is designed to prevent undesirable air turbulence, which could cause particles to become airborne. To accomplish this, cut-outs 20 are located near the sub-assemblies. This elim- inates an area for particle accumulation on mounting plate 19 and allows an undisturbed downward flow of clean air through top mounting plate 19.

- - Robot unit 11 is designed to operate in a class 10 clean room environment. Multi-gripper 12 connects to the arm of robot unit 11 to perform two tasks: to manipulate the carrier and to manipulate wafer 21. Pneumatic air operates the multi-gripper 12 so that fingers attached to each side of the gripper can grasp and secure the carrier (see the preceding article). Another finger, on multi-gripper 12, secures wafer 21 by means of a vacuum attachment. The vacuum attachment allows random access to the wafers from their cassette nests 13. All of the processing sub-assemblies are strategically mounted on top plate 19 so that robot unit 11 can maneuver without having to violate the air space over the wafers, or sub-assemblies.

- - A potentiometer position sensing unit (not shown) is used to sense the table movement and an elevator location. Information is sent to robot 11 prior to placing a carrier onto an elevator for further processing.

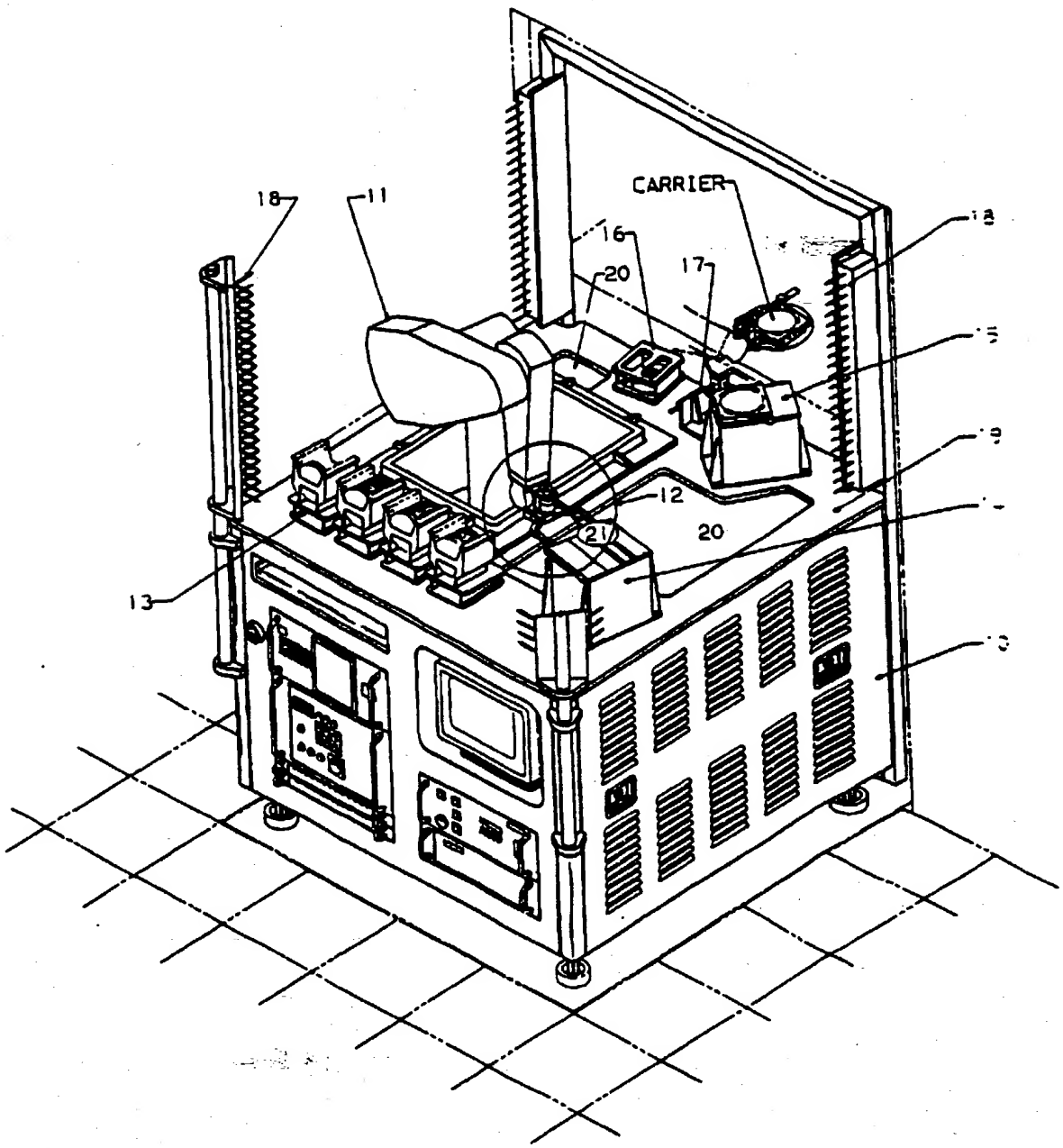
- - By using four cassette nests 13, loading time is reduced. The nests which hold the wafer cassettes are elevated to aid in reducing air turbulence. The nests hold the wafers in a vertical position rather than the traditional horizontal position. This combined with a large cut-out at the bottom of the nest allows a continuous stream of clean air to flow between the wafers, constantly washing any

particles from their surfaces.

- - Aligner 14 is totally enclosed with an internal vacuum attachment so as to control and remove any particles produced by the mechanical mechanisms. Carrier spreader 15 automatically positions a wafer 21 and opens and closes the carrier. Like aligner 14, carrier spreader 15 also is totally enclosed with an internal vacuum attachment to control and remove any particles produced by the mechanical mechanisms. Idle stand 16 is a resting device for the carrier. X, Y, Z-sensing device 17 is mounted on top plate 19 and relays information back to robot 11 as to the location of the elevator. Like cassette nests 13, it is also designed to allow air to flow through its center and down through a cut-out in the mounting plate. Light curtain 18 is a safety device used to protect the operator.

- - Hard-coated aluminum is used to prevent particle production from oxidation. Polished stainless steel is used to prevent particles from sticking to surfaces. The underside of top plate 19 is painted with epoxy.

- - The controls and the software used with the system are designe with the user in mind. The controls, located under mounting plate 19, are on slides to allow easy access. The software contains a menu-driven diagnostic program, a maintenance program, a recovery program and an operator instruction work status program.


18
11
CARRIER
16
20
17
12
21
20
13

XP-002083222

AN : NN9105251

PUB : IBM Technical Disclosure Bulletin, May 1991, US

VOL : 33

NR : 12

PG : 251 - 259

TI : High-Speed Opens And Shorts Substrate Tester.

TXT : - Described is an assembly of custom-designed ceramic substrate handlers and fixtures which are combined with commercial devices to test substrates for electrical opens and shorts at a high speed.

- - In prior art, the testing of substrates for opens and shorts, as used in circuit boards and cards, required manual loading and unloading one part at a time and test probes were positioned manually. The concept described herein eliminates the prior-art manual effort and provides automatic handling and testing operations of the substrates. This not only speeds the testing operations, but improves the reliability of the tests. The five figures show the various functional mechanisms which make up the assembly. Fig. 1 shows the overall substrate tester. Fig. 2 shows an exploded view of the first section of the tester and Fig. 3 shows an exploded view (section A-A of Fig. 2) of typical probe testing station 16. Fig. 4 shows an exploded view of the output handler and Fig. 5 shows the details of the transfer mechanisms.

- - The overall substrate tester, as shown in Fig. 1, includes input handler 1, which comprises of two input elevator drawers 4, input robot 2, and input conveyor belt 3, also shown in Fig. 2. To load parts, the operator slides out drawer 4 enabling wire basket 5, which is full of trays, to be loaded into the inside section of drawer 4. Wire basket 5 can hold up to ten trays and each tray can hold up to twenty-five substrates. An empty wire basket 5 is loaded into the outside section of drawer 4. Once the loading operation is completed, the operator slides drawer 4 back into the machine and repeats the loading operation with the other drawer. The input of the tester has now been loaded with a maximum of 500 substrates.

- - Handler control computer 6 operates under software control from sensors when input drawers 4 are closed. Handler control computer 6 turns on a pair of solenoids (not shown) which pressurizes two pairs of cylinders (not shown) which move up through closed drawers 4 and lifts up two full stacks of trays from wire basket 5. The stack of trays are stopped by a set of tray clamping fingers (not shown) above each drawer 4. In this way the top tray of substrates is accurately positioned against a top surface as well as in the X/Y plane for robotic loading into the rest of the tester.

- - Input robot 2 picks five substrates at a time from the tray and places them into pockets of input conveyor belt 3. Belt 3 is indexed forward until a substrate is sensed by a sensor located at shuttle 10 (Figs. 1 and 2). Input conveyor belt 3 then waits for shuttle 10 to "rake" the substrate off input conveyor belt 3 and into a mid-tool shuttle track (not shown). After input conveyor belt 3 has indexed

five times, input robot 2 will do another pick-and-place cycle. After five pick-and-place cycles, the top tray in drawer 4 will be empty of substrates. Input robot 2 will then pick up the top empty tray and place it into the side of drawer 4 where the empty basket had been loaded. Input robot 2 can now repeat the next five pick-and-place substrate cycles, after which it will repeat the tray pick-and-place cycle. Eventually, the once empty drawer 4 will have ten empty trays and drawer 4 that once was full will now be empty of trays. Once this happens, input robot 2 will automatically shift to unloading substrates out of a second drawer 4, which also has a stack of ten full trays. While input robot 2 is unloading the second drawer 4, the operator can refill the first drawer 4 without having to interrupt the operation of the tester.

- - At the middle of the tester, called mid-tool, are four separate stations; substrate serial number read station 7, substrate chip site mapping station 8 and two identical testing stations 9a and 9b. Shuttle 10 consists of a walking beam which moves the substrates through mid-tool. The program controlling shuttle 10 will not index it until a substrate is present at the input conveyer intersection. This way, all of the nest positions of shuttle 10 will always be filled with a substrate. Shuttle 10 slides all of the substrates forward in the mid-tool track simultaneously, then raises up above the substrates and retracts to its original position, then lowers itself over the indexed substrates.

- - At each station, there is a four-point locating jaw 11 (Figs. 1 and 2) that is mechanically timed with the indexing motion. Jaws 11 close on the substrate just as shuttle 10 finishes sliding forward, but before shuttle 10 lifts up. Jaws 11 open just prior to the forward sliding motion of shuttle 10. Both the motion of shuttle 10 and the clamping motion in the mid-tool are mechanically tied together by a single hydraulic motor (not shown).

- - All mid-tool stations operate in parallel on the substrates in their respective stations. Read station 7 contains moveable camera 12 (Figs. 1 and 2), which is connected to optical character recognition (OCR) system 13 so as to read the serial number on the substrate. OCR system 13 then communicates the serial number on the substrate to handler control computer 6. Manufacturing software running along with tool control software can now track individual substrates and attach mapping and test information to the serial number.

- - Substrate mapping station 8 contains a camera (not shown) positioned on an X, Y, Z stage 29 (Figs. 1 and 2), so that movement can occur across all of the chip sites of the substrate. A Z-axis motor (not shown) moves the camera for the purpose of focusing on the different substrate thicknesses, which can vary. The camera on X, Y, Z stage 29 is connected to vision system 14, which is designed to measure the actual dimensional positions of the chip sites from their nominal positions. The dimensional data is communicated to handler control computer 6 where the data is stored with the serial number of the substrate. This dimensional information, called mapping data, will be communicated to test station's probe positioning controller 15 prior to the substrate being indexed into first test station 9a. Here, probes are moved by the motion of a motor-driven step cam and cam follower (not shown). Probe-positioning assembly 16, as shown in Fig. 3, consists of a four-axis positioning system (not shown) and has X, Y, theta and quadrature motions. The quad axis is necessary for a four-chip site substrate and is not used for a single chip site substrate. The quad axis moves the separate four-probe cluster radially in or out from the center of the cluster. The X, Y, and

theta axis moves the probe cluster to the "best fit" position that has been calculated from the mapping data.

- - Once the substrate has been four-point located and the probes have been positioned by the X, Y, theta and quad stages, probe press 17 (Figs. 1 and 2) is lowered so that probes will contact the test points at the top of the substrate. Probe press 17 is cam driven by a hydraulic motor (not shown) so as to obtain the speeds and forces required by the tester. The underside of the substrate also has electrical test points that are contacted by fixed position spring probes (not shown). Probe positioning assembly 16 uses buckling beam type of probes with a maximum of 2,304 individual buckling beams within a probe assembly. There are 640 spring probes contacting the underside of either the four- or single-chip site size substrate. The spring probes and the buckling beam probes are wired back to high-speed test matrix 18a and 18b, which measures the electrical resistance of the known electrical paths through the substrate. If the resistance of a path is above a certain value, then the path is considered an electrical open. If the electrical resistance between pairs of paths are below a certain value, then the paths are considered shorted. The test data is communicated to test data tool control computer 27 and then is passed to test analyzer 19 by means of a local communications controller (LCC) (not shown). Tool control computer 27 analyzes the data and classifies the substrate as accept, reject, rework or retest. This test data can also be attached to the substrate serial number.
- - When the test is complete, the test probes are raised so that shuttle 10 can index the substrate. Shuttle 10 is mechanically and electrically interlocked with the probe's up and down position so that shuttle 10 cannot index the substrate while the probes are in the down position. The substrates will be indexed from test station 19a and 19b which are identical to each other. The same test will be repeated at test station 9b as was done at test station 9a. The results are compared, such that if the two tests differ, then retesting of the substrate may be indicated, or a repair and/or a recalculation of the test probes may be required. This redundant test improves the reliability of the substrate test as well as being a quality test of the buckling beam probes themselves.
- - From test station 9b, the substrates are indexed by shuttle 10 from the mid-tool shuttle track (not shown) to sort conveyor intersection output handler 25, also shown in Fig. 4. Output handler 25 consists of sort conveyor 20, five transfer mechanisms 21 (detailed in Fig. 5), five output conveyors 22, output robot 23 and six output elevator/drawers 24.
- - Shuttle 10 slides the substrates from the last position of the mid-tool shuttle track (not shown) into a pocket of sort conveyor 20 on the forward stroke. Once the shuttle is in the retracted position and handler control computer 6 senses a substrate present at the intersection, then sort conveyor 20 will be commanded to index the conveyor belt forward one position and then it stops. In this way, all pockets of sort conveyor 20 are kept full.
- - Sort conveyor 20 intersects with five output conveyors 22 at five different locations. The five output conveyors 22 are called Good 1, Good 2, Reject, Rework and Retest conveyors. Handler control computer 6, which has been indexing shuttle 10 and sort conveyor 20, can count the number of indexes so that it knows when a certain tested substrate should be at a particular intersection on sort conveyor 20. There are also "Part Present" sensors at each intersection to confirm the presence of the expected substrate. The tested and categorized substrate will be indexed along on sort

conveyor 20 until it reaches the intersection corresponding to its tested classification.

- - When the expected substrate is at the proper intersection and sort conveyor 20 is stopped, then handler control computer 6 can command transfer mechanism 21 to retract from its normal extended position. This causes the substrate to be transferred from the pocket of sort conveyor 20 to the first pocket on output conveyor 22. A transfer mechanism (not shown) is then extended out to wait for its next substrate. Output conveyor 22, which has just received the next substrate, is indexed one position and stops and waits for the transfer of the next substrate. This assures that all pockets of output conveyor 22 will have a substrate in it.

- - The last pocket position in output conveyor 22 has a substrate present sensor to signal when it cannot be indexed any more without first commanding output robot 23 to do a pick-and-place unloading operation. Output robot 23 is identical to input robot 2, except that it has a longer travel in one axis for the purpose of unloading six possible different elevator/drawers 24. Output elevator/drawers 24 have the same names as output conveyors 22, i.e., Good 1, Good 2, Reject, etc. Plus, there is a sixth spare elevator/drawer available.

- - Output elevator/drawers 24 are identical to the input drawers 4. The operator loads wire basket 5 full of empty trays into the outside section of the elevator and an empty wire basket with no trays into the inside section. When elevator/drawer 24 is closed and the two elevator halves have been raised, output robot 23 will pick the top empty tray from the outside stack and place it on the inside elevator. Output robot 23 will then be able to do pick-and-place operations from output conveyor 22 corresponding to the output elevator. When the tray is filled with substrates, five pick-and-place operations, output robot 23 will then take the next empty tray and place it on top of the tray that it has just filled. Eventually, the inside elevator will have ten trays that are filled with tested substrates. Sensors in elevator/ drawer 24 will tell handler control computer 6 when inside elevator/ drawers 24 is full or when the output drawer is empty of trays. Handler control computer 6 will then signal the operator via an indicator light above the elevator or display a message on one of the computer terminals. Since most tested substrates are expected to test as Good parts, two Good output elevators (Good 1 & Good 2) are designed into output handler 25 so that robot 2 can switch unloading from one to the other without having to stop the operation of the tester when one elevator is filled.

- - The section of the tester which controls the operation consist: of the following units: three of AC/DC power distribution cabinets 26; three computers - 6, 27, and 19; four computer terminal/printer half racks 30; two cabinets of motor position controller 15; two pairs of cabinets for test matrix/computer 18a and 18b; vision system 14, serial reader and conveyor cabinet assembly 13 and two interface assemblies 28, which are housed in two of the half racks 30. The three power distribution cabinets 26 contain all the primary and secondary power on/off controls, main and branch circuit breakers, branch circuit isolation transformers, AC line filters and DC power supplies.

- - The three computers are required for the tester and run specially designed programs. Handler control computer 6 controls all the operational functions of the tool. This includes all sensor and actuator digital-in and digital-out (DI/DO) control and all communications to the other intelligent equipment. Tool control computer 27 operates at one level higher in the computer control

architecture, similar to an area controller, and talks down to handler control computer 6. It also communicates to test computer/analyzer 19 so as to obtain the test results. Its programming allows it to operate in a computer integrated manufacturing environment. Computer/analyzer 19 processes and tracks the job lot through the testing operation and receives test data from tool control computer 27 and processes the data. Each computer has a terminal and printer as input/output (I/O) devices, except handler control computer 6 which has an extra terminal. All operator control and maintenance functions are handled through software menus on the terminals of handler control computer 6.

- - Two motor control cabinets house positioning controller 15 for controlling all (electric and hydraulic) motors, except the seven conveyor motors. Handler control computer 6 communicates with the motor control cabinets across four communication lines, depending on which motor it wishes to "talk to". Handler control computer 6 commands position controller 15 via DI/DO signals, i.e., open/close, extend/ retract, etc. The cabinets which house high-speed test matrix/computer 18a and 18b perform the actual opens and shorts electrical testing. One cabinet houses the test matrix while the other houses the computer that controls the test matrix.
- - Vision system 14 maps the chip site test points on the substrate. It communicates the mapping coordinate information to handler control computer 6 and controls the X, Y, Z, axis of the map station motors. The cabinet which houses optical code recognition (OCR) system 13 for reading the substrate numbers also contains conveyor motor control assembly (not shown). OCR system 13 communicates to handler control computer 6 which, in turn, controls the DI/DO commands to the conveyor motor control assembly.
- - Cabinet interface assemblies 28 reside side by side in two half- high 19" racks and interface all of the handler computer 6 DI/DO signals to and from the main tester as well as the other cabinets. One of the interface assemblies also contains an anti-crash unit and conveyor control and safety logic card (not shown). This multi-function logic card uses combinational hardware logic so as to electrically interlock conveyor indexing functions for operator safety.
- - The tester has the ability to handle several different sizes of substrates. To change the tester to handle different substrate sizes requires the following: change substrate trays of wire basket 5; change input conveyor assembly 3, which unplugs and slides off input handler 1; change shuttle 10 and its track; change X, Y, Theta, Quad axis and probe position assemblies 16 (Fig. 3); change sort/output conveyor 20 and 22, which unplugs and slides off on one large plate from output handler 25.

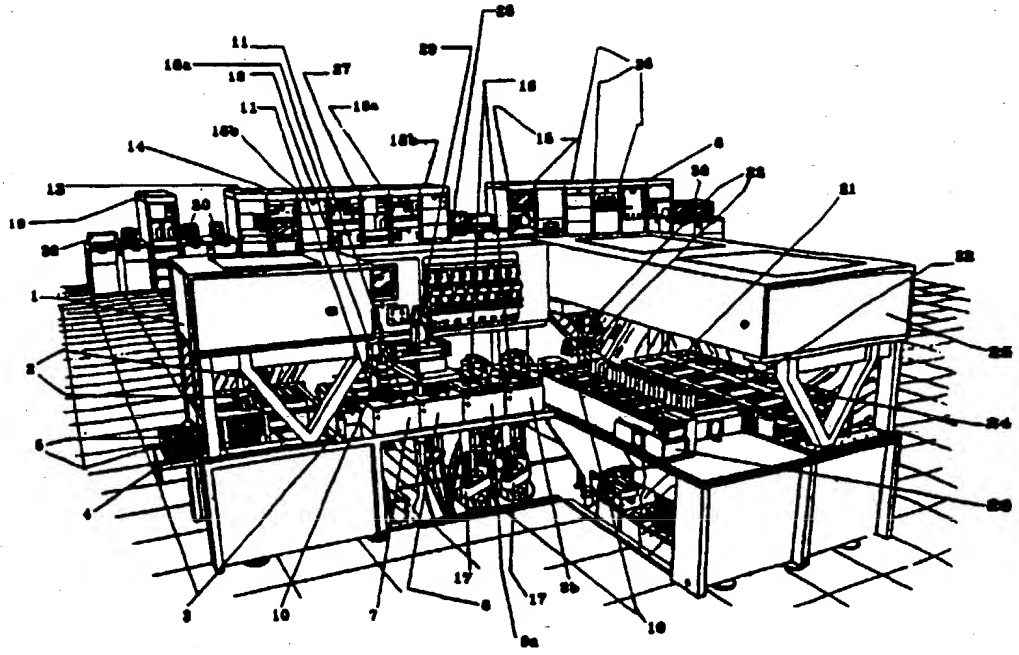


FIG. 1

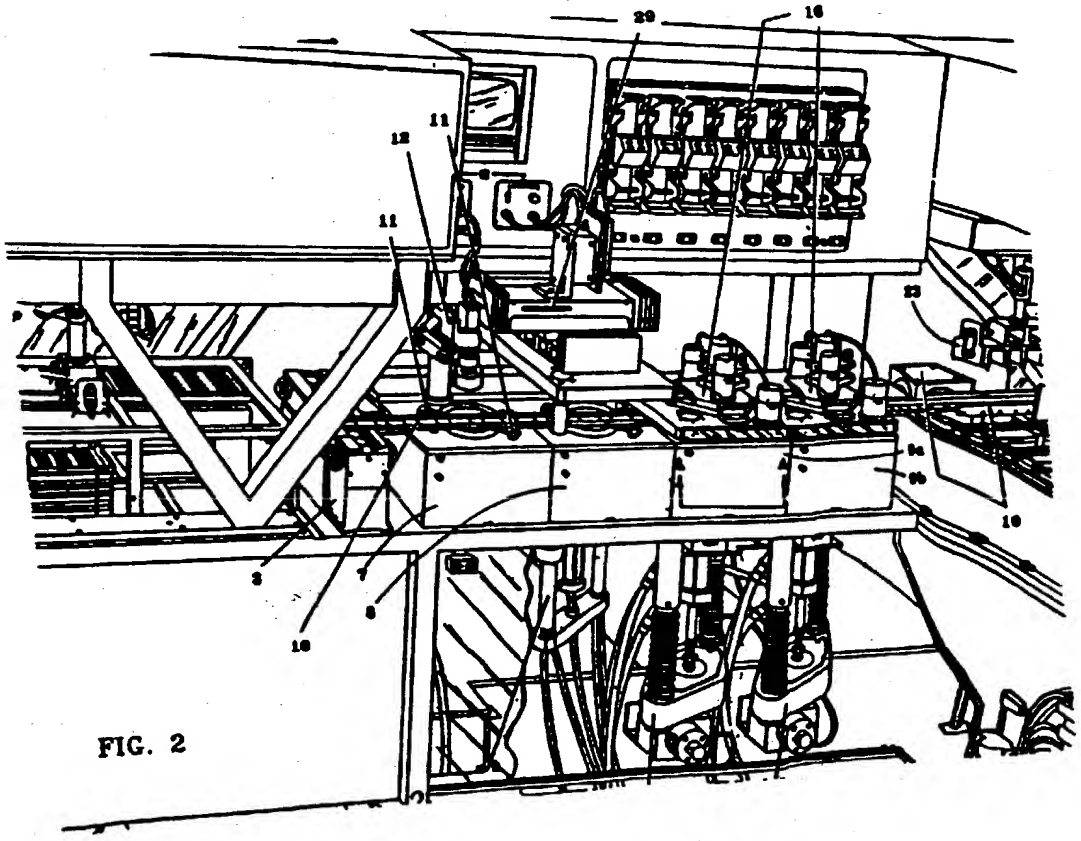
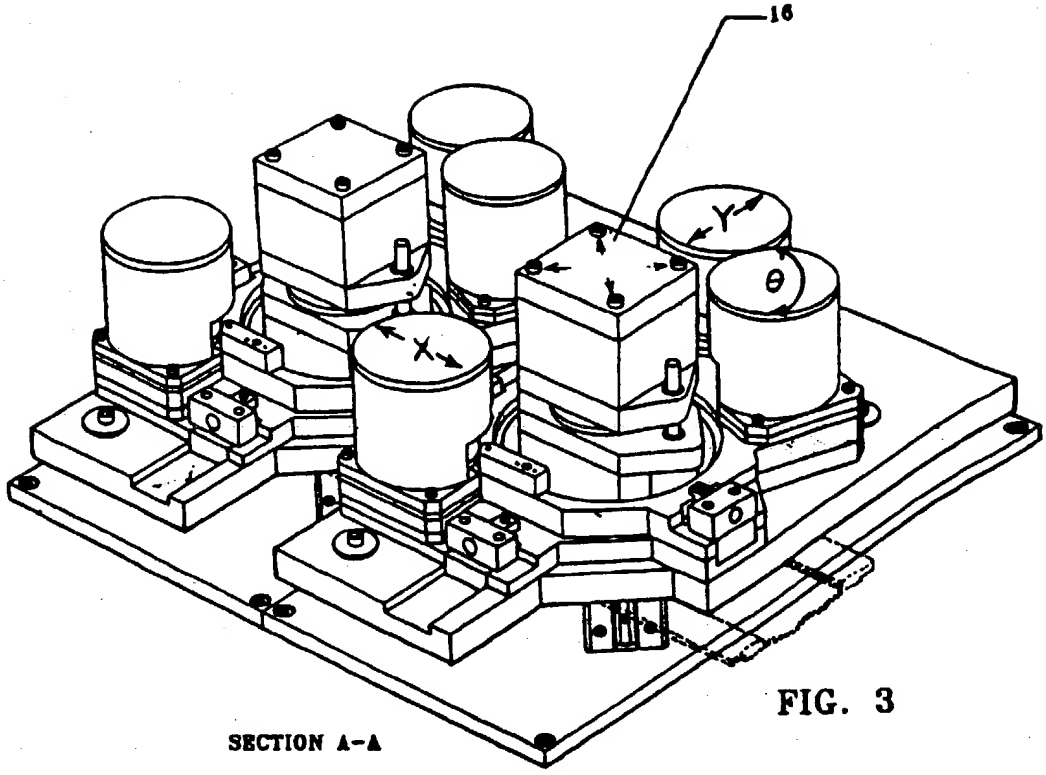


FIG. 2



SECTION A-A

FIG. 3

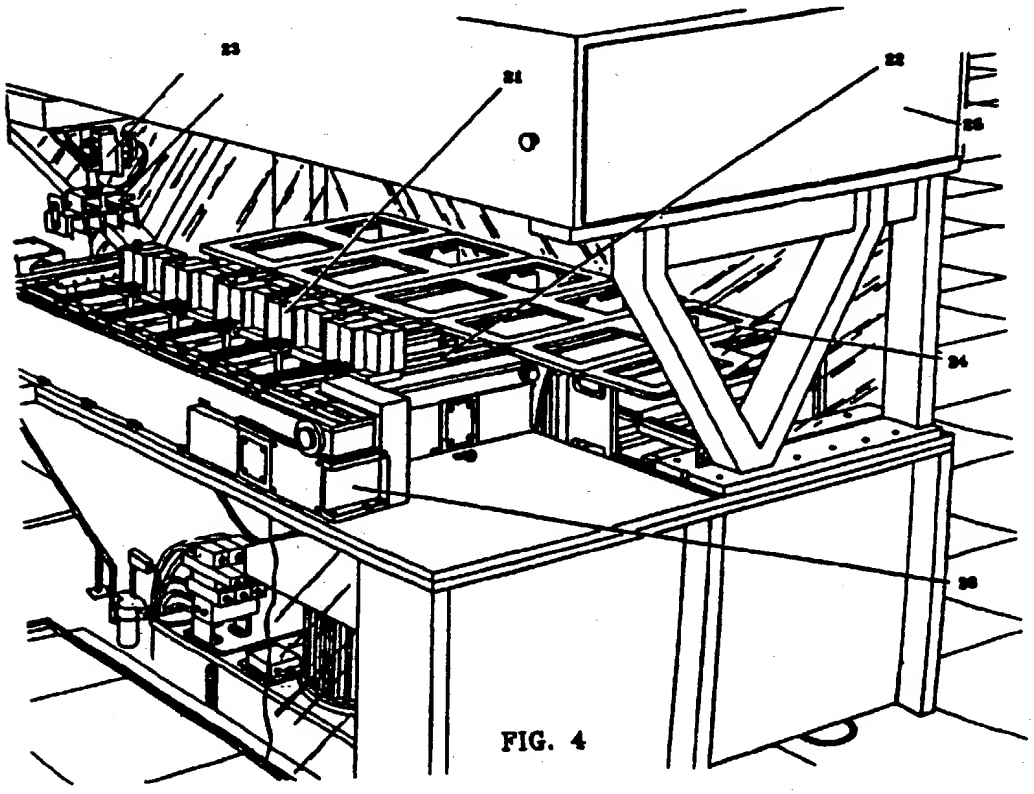


FIG. 4

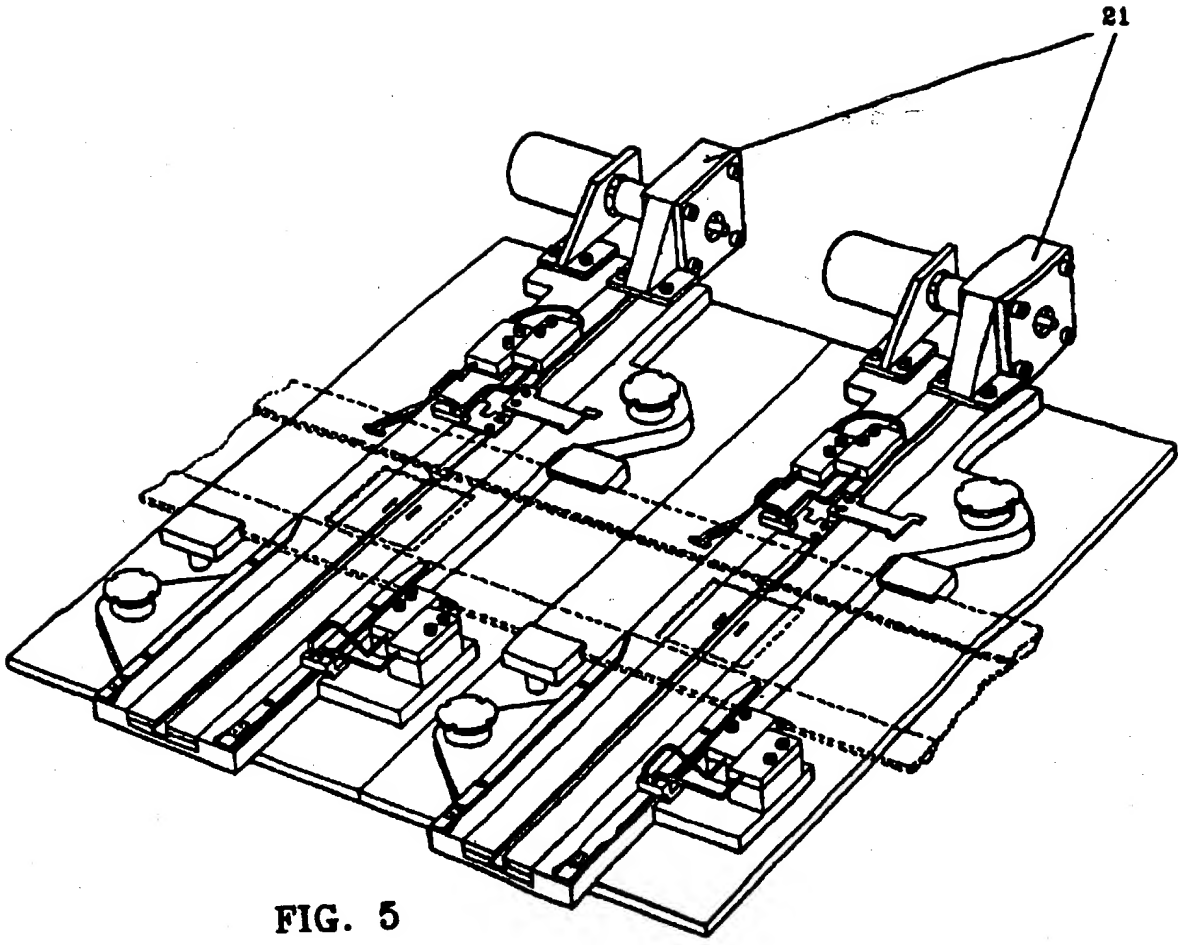


FIG. 5